JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming

IFF Spring School 2014

March 13th 2014 | Florian Janetzko, Daniel Rohe and Alexander Schnurpfeil

JÜLICH
FORSCHUNGSZENTRUM

… or:

DON‘T PANIC!

and CARRY a DEBUGGER

A scientist's quick-guide to parallel programming

Florian Janetzko, Daniel Rohe and Alexander Schnurpfeil

IFF Spring School 2014

Reminder: Why Parallel Programming?

The clockspeed of current elementary computing units does not increase anymore. Instead, the units themselves grow in number, at all scales. (Moores law is still valid.)

→ The full potential of such systems can only be exploited by using highly and hierarchically parallelised code.

Examples

JUQUEEN

Forschungszentrum Jülich
No. 8 in TOP 500 , Nov. 2013

- 458,752 cores at 1.6 GHz
- 28,672 nodes
- 16 cores per node
- 4-way double precision SIMD unit per core

In either type of environment, parallelisation can be (and often is) based on the same software standards, namely on MPI and OpenMP.

Office cluster

Anywhere in the world

- 24 cores
- 12 nodes
- 2 cores per node
- 2-way double precision SIMD unit per core

Outline

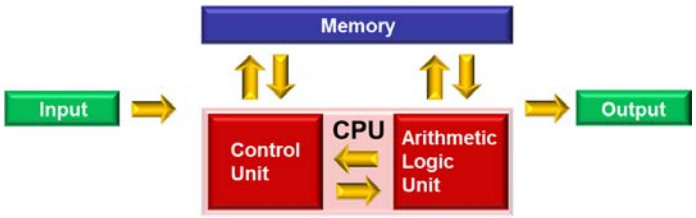
1. Parallelising an algorithm – a real-life non-computing example
2. OpenMP: An example and some hints
3. MPI at a glance
4. *… some thoughts …*
5. Where to go from here?

Outline

1. Parallelising an algorithm – a real-life non-computing example
2. OpenMP: An example and some hints
3. MPI at a glance
4. *… some thoughts …*
5. Where to go from here?

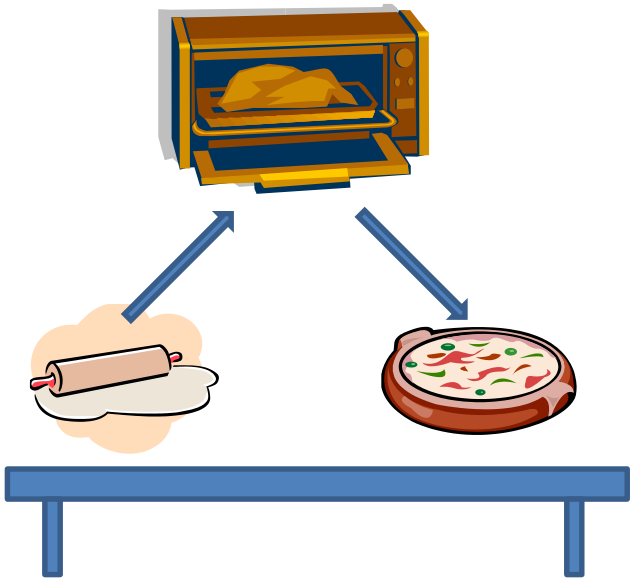
Computing vs. Real Life

von Neumann architecture



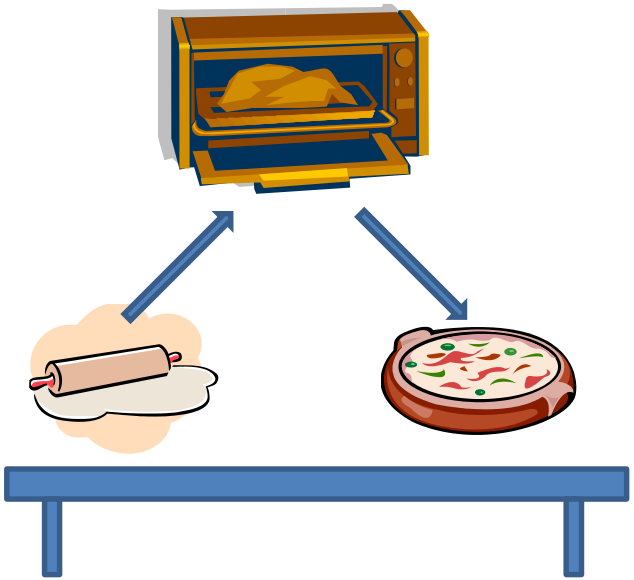
→ In the simplest case one instruction is applied to one element of data with each cycle.

Single instruction, single data

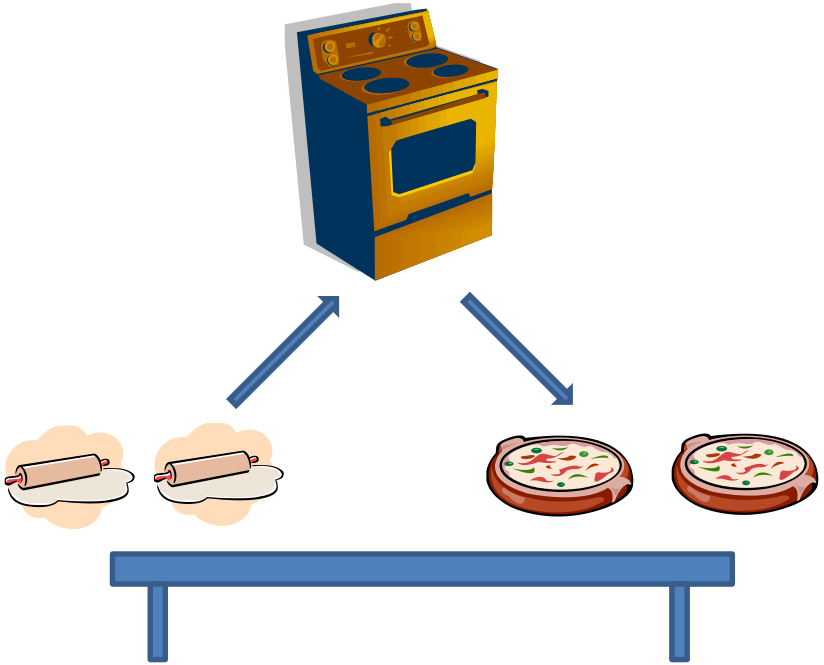


From SISD to SIMD

Single instruction, single data

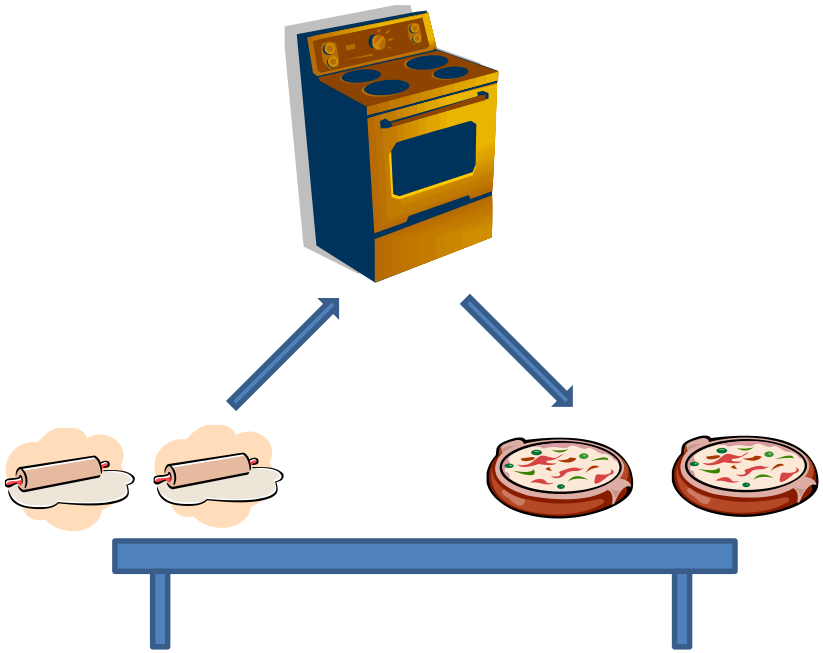


Single instruction, multiple data

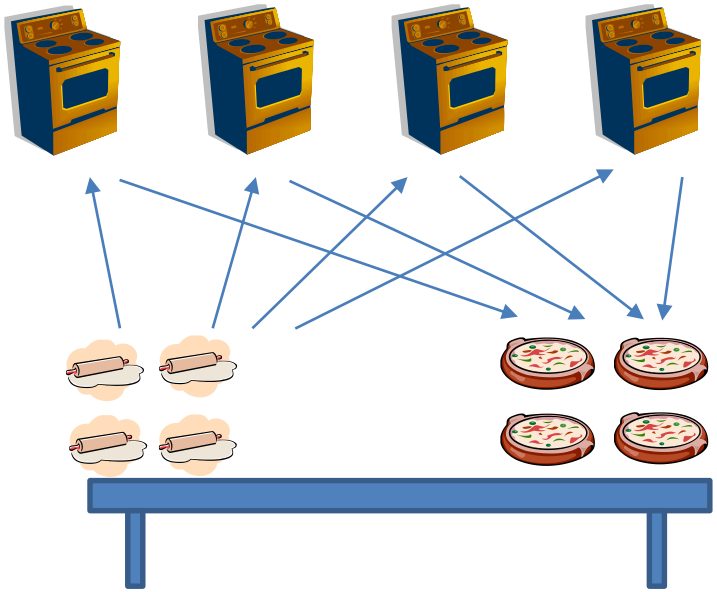


From Single-core to Multi-core

Single instruction, multiple data

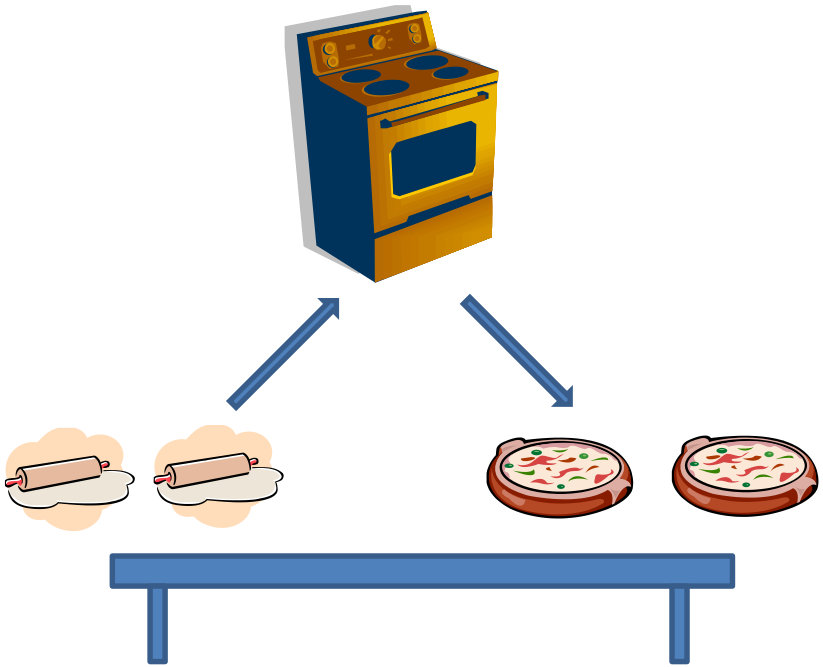


Multi-core

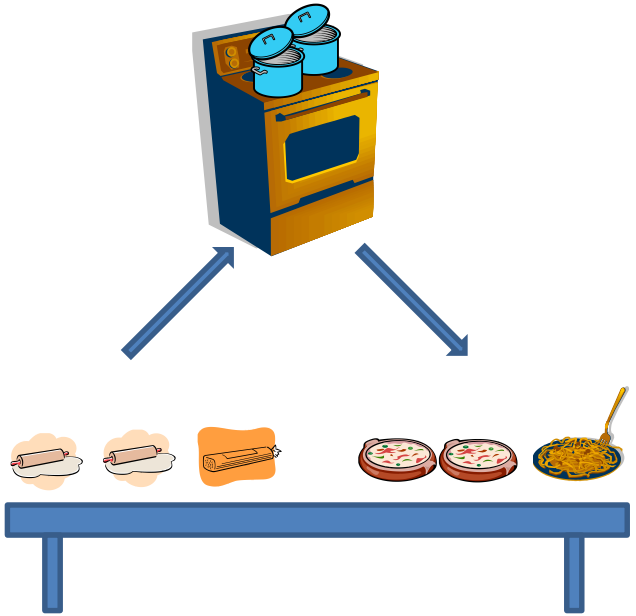


From SIMD to MIMD

Single instruction, multiple data

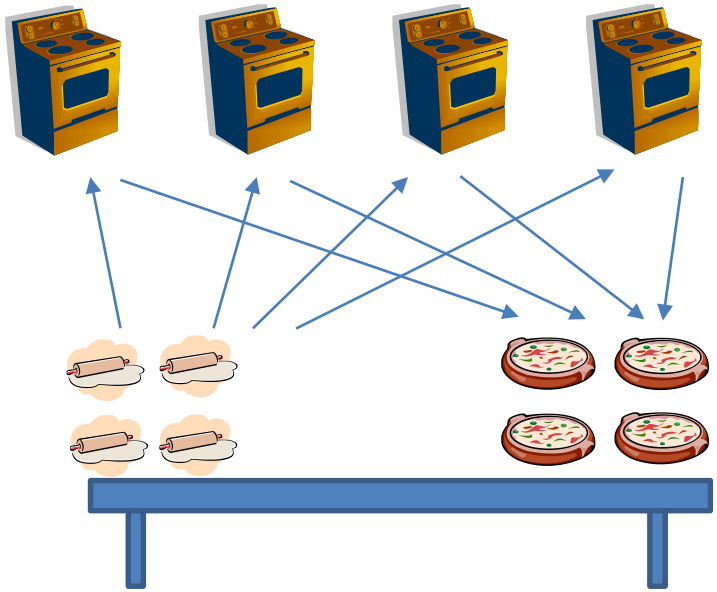


Multiple instruction, multiple data

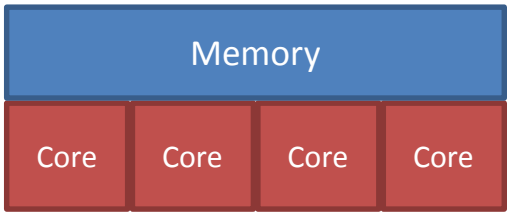


Real Life vs. Computing

Multi-core



Multi-core shared memory



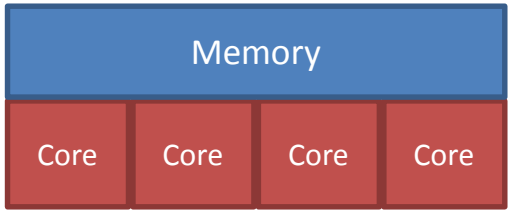
→ OpenMP-ready

Outline

1. Parallelising an algorithm – a real-life non-computing example
2. OpenMP: An example and some hints
3. MPI at a glance
4. *… some thoughts …*
5. Where to go from here?

Code Example

```
for i=0 to i=999.999
{ y[i] = exp(x[i]); }
```

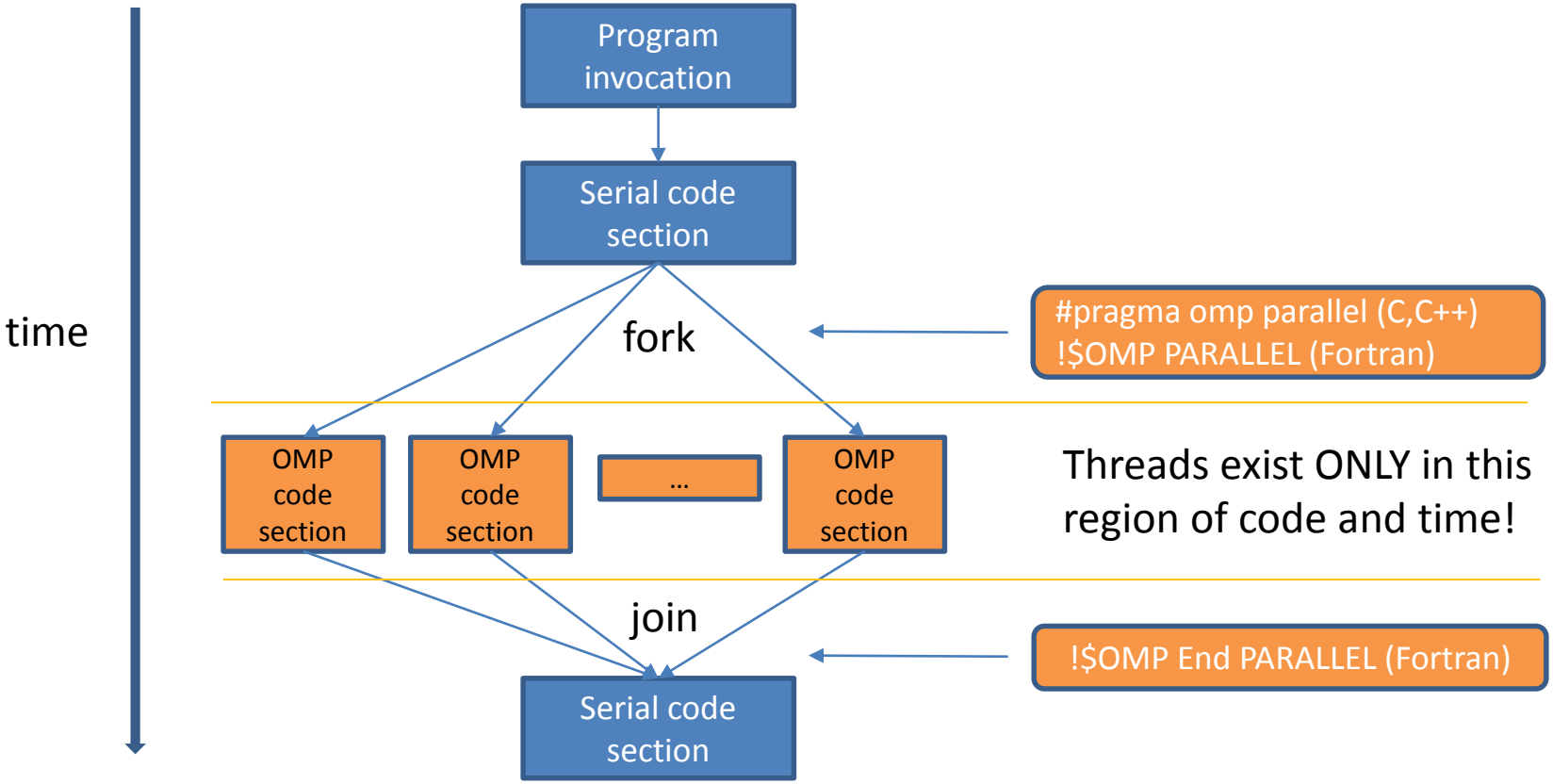


Code Example: OpenMP in C/C++

```
some serial code
…
omp_set_num_threads(N);          // setting the number of threads to be created
…
#pragma omp parallel             // only at this point the threads will be created (!)
{
#pragma omp for schedule(static) // workload distribution defined a priori using ‚static‘
for(int i=0;i<=999999;i++)
{ y[i] = exp(x[i]); }

}
```

OpenMP – Thread Lifecycle



Controlling OpenMP

```
…
omp_set_num_threads(N);          // setting the number of threads to be created
…
#pragma omp parallel             // only at this point the threads will be created (!)
{
```

OpenMP offers a variety of ways how the threads, which are available in this region, can be used to share workload.

```
}
```

OpenMP Caveats: Overhead

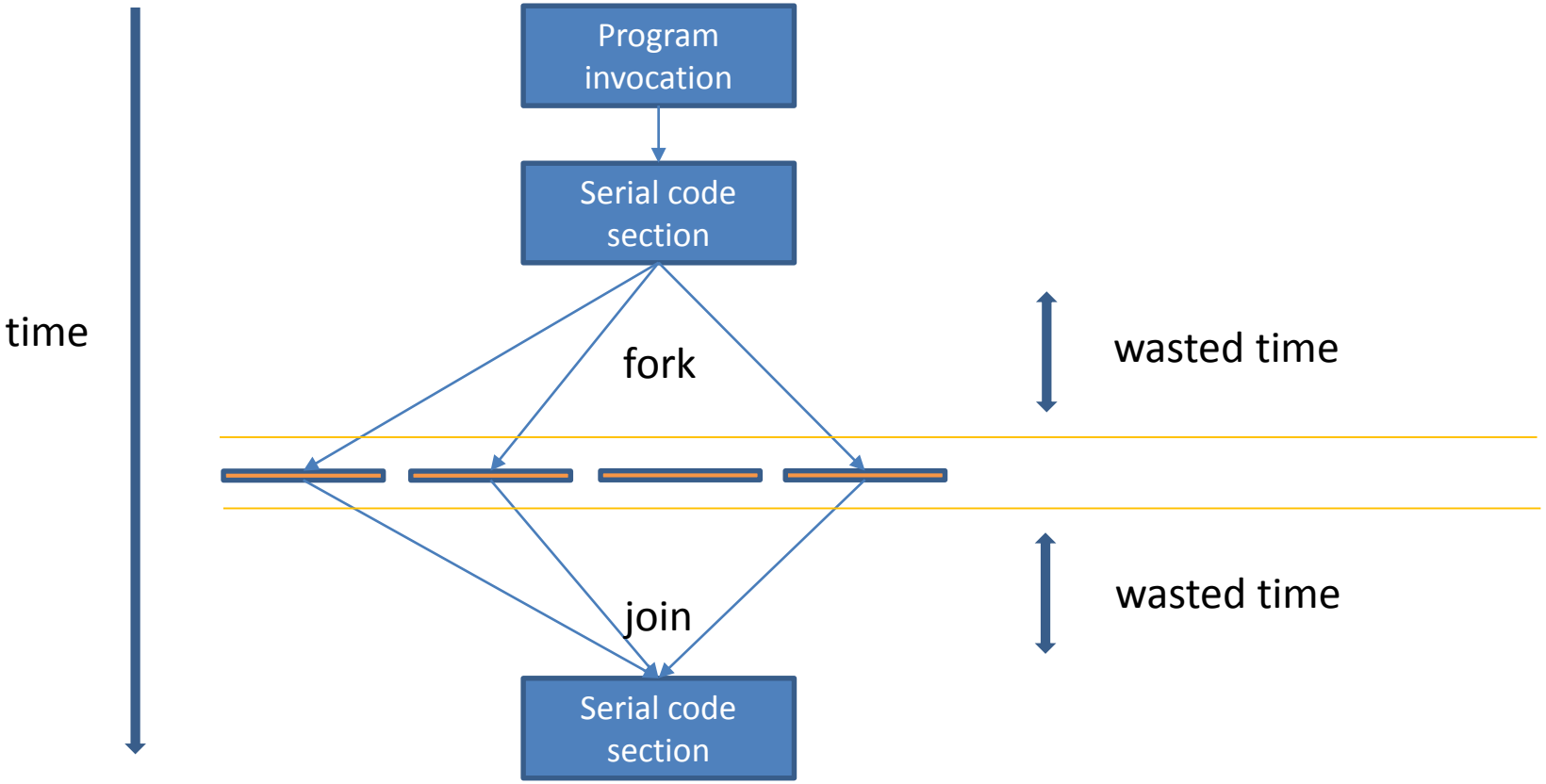
OpenMP logistics may spoil any performance gain completely:

```
for i=1 to i=10
{ y[i] = exp(x[i]); }
```

Parallelising this with OpenMP would be unreasonable.

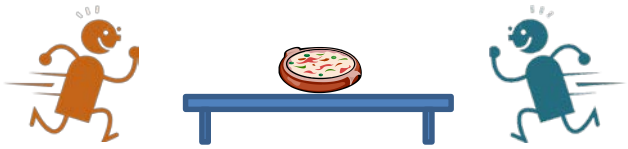
→ The influence of „logistical overhead“ needs to be analysed when parallelising a code with OpenMP.

OpenMP Caveats: Overhead



OpenMP Caveats: Race Conditions

Shared memory: All threads have access to the same data → Too many cooks spoil the broth!



Two threads may write to the same address in the wrong order. This is called a „race condition“ and it can (and most likely will) mess things up badly while not disturbing the technical execution of the code at all. (READING from the same address is ok, though.)

This is THE MOST common and important issue that needs to be avoided when using OpenMP.

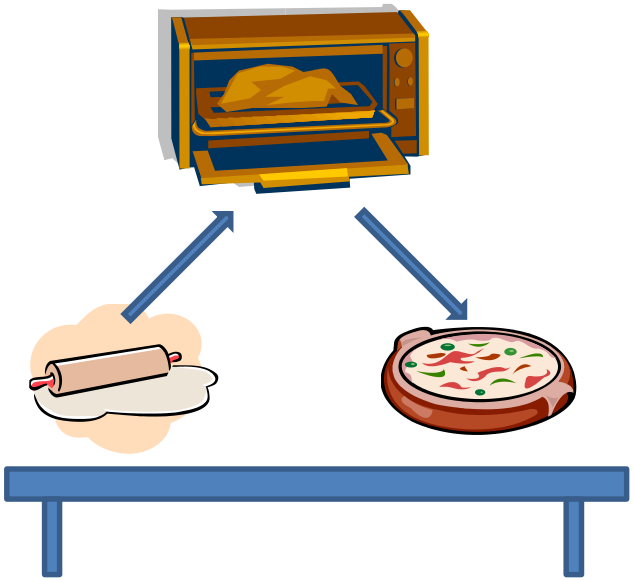
Therefore any code using OpenMP MUST be run through an appropriate tool which can chase up such race conditions.

Outline

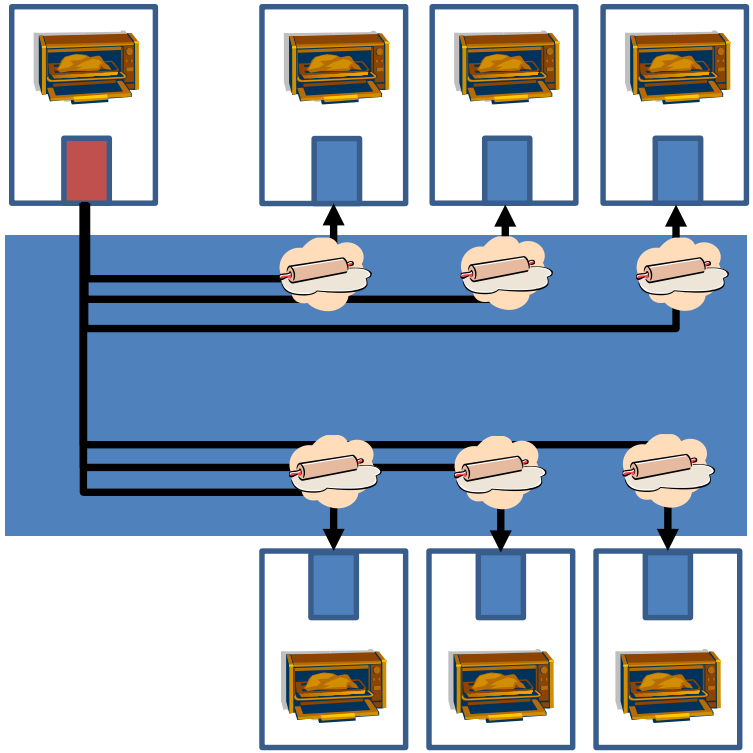
1. Parallelising an algorithm – a real-life non-computing example
2. OpenMP: An example and some hints
3. MPI at a glance
4. *… some thoughts …*
5. Where to go from here?

From SISD to Distributed Computing

Single instruction, single data

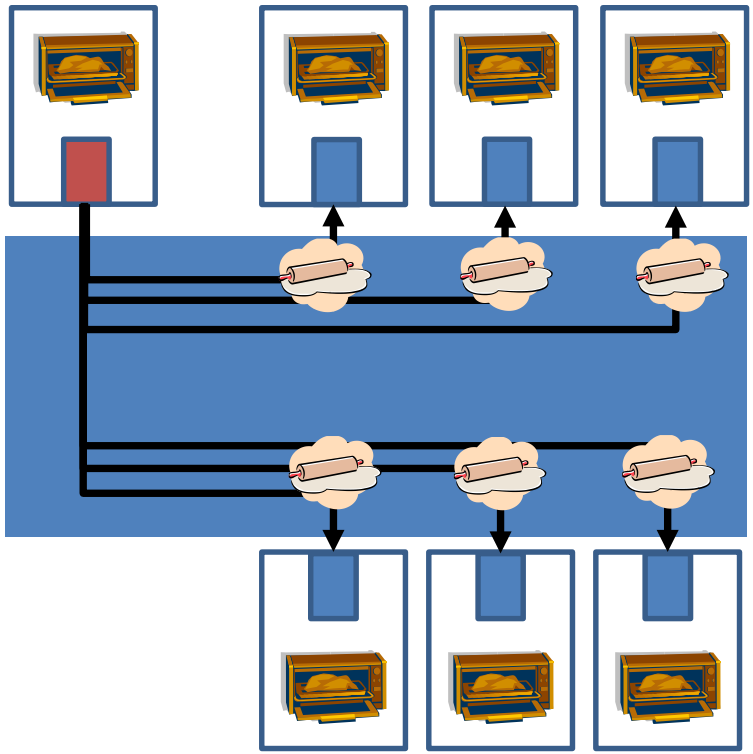


Distributing the workload

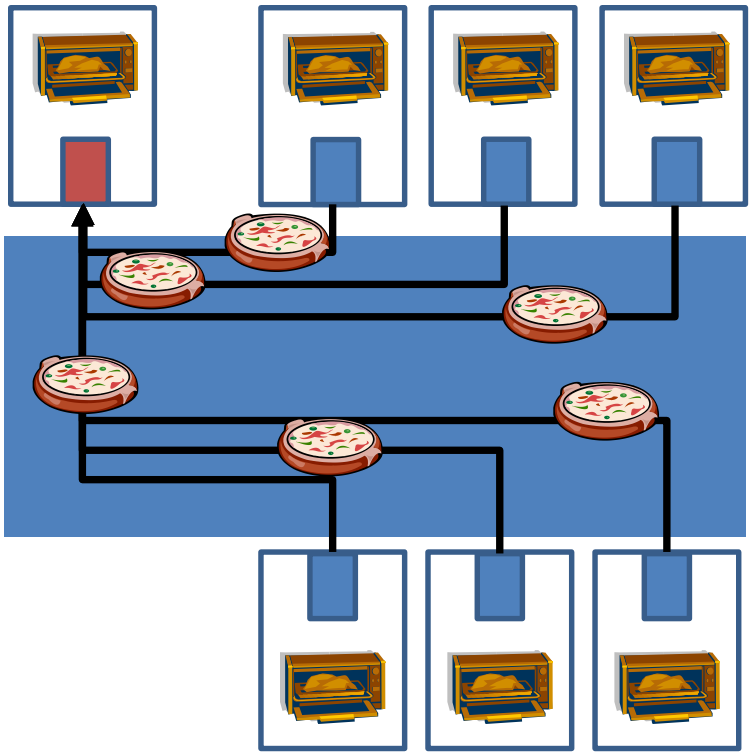


From SISD to Distributed Computing

Distributing the workload



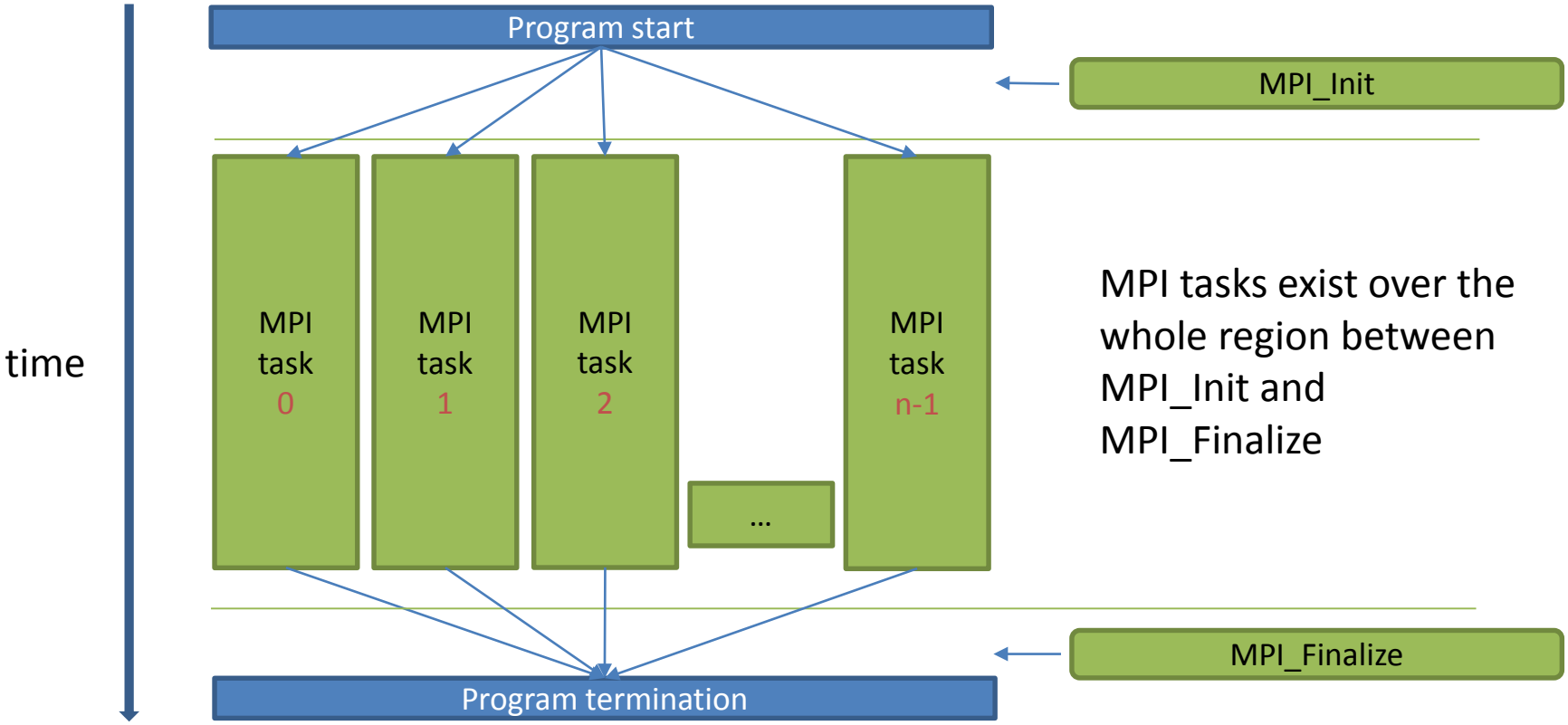
Collecting the results



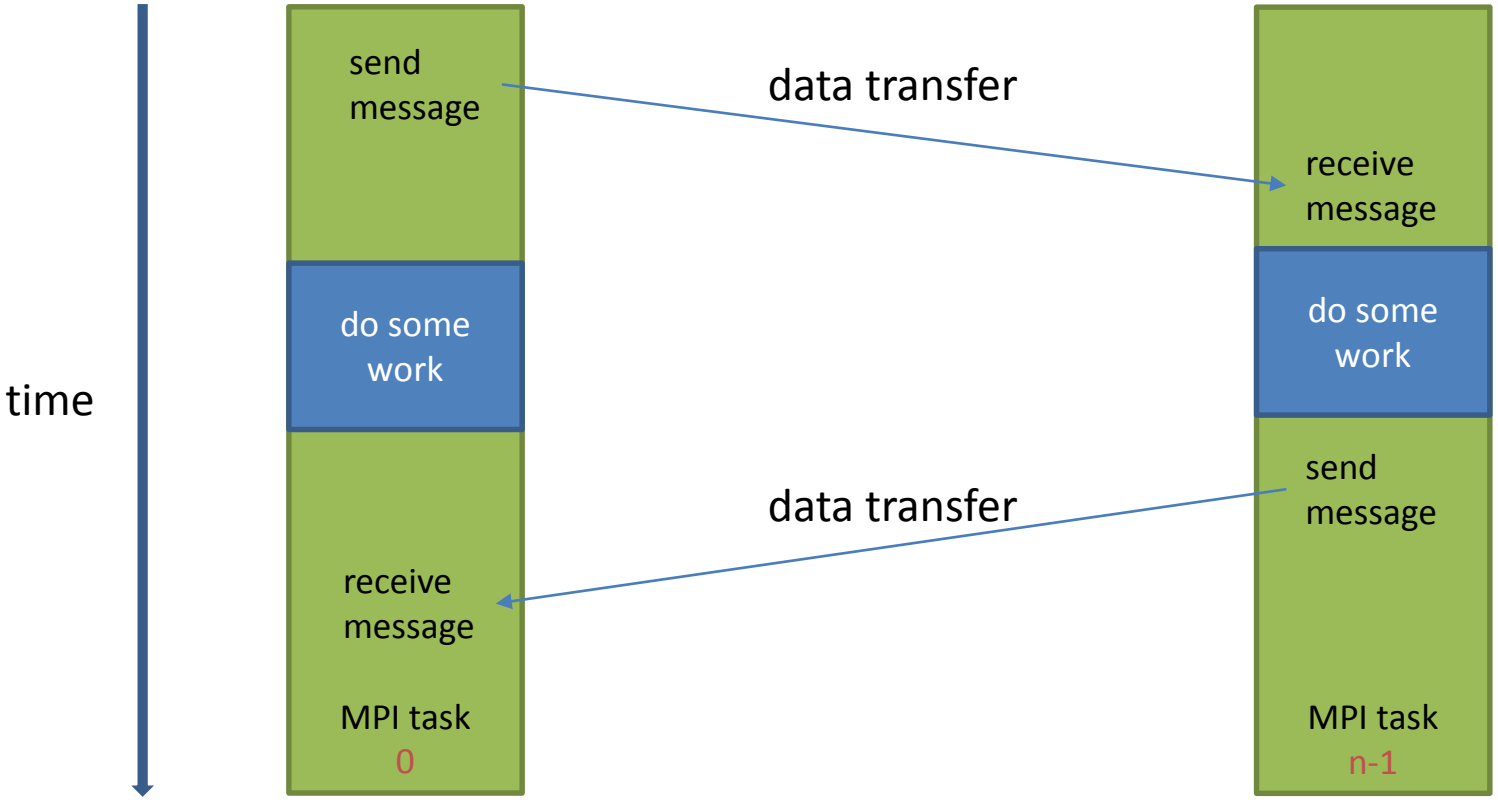
Message Passing Interface - MPI

1. MPI allows to exchange messages between separate instances of single or multiple applications, called tasks.
2. MPI tasks can run on the same node, on separate nodes, or even on spacially completely separated machines.
3. Each task always has it‘s own memory space (even when running on the same node!).
4. Workload logistics and data exchange needs to be done by the developer.

MPI – Task Lifecycle



MPI – the very principle



MPI: Some Code Sketching

```
MPI_Init(…);
…

if ( 0 == my_task_id )                                  // the master distributes the raw data
{
for( task = 1 ; task < total_number_of_tasks - 1 ; task++ )
   { MPI_Send( &x[0]  , 1000000 , MPI_DOUBLE , task     , ... ); }
                  source , number   ,  type             , who to
}

else                                                    // the tasks receive the raw data
{
  { MPI_Recv( &x[0]  , 1000000 , MPI_DOUBLE , 0          , ... ); }
                  target , number   , type              , who from
}

…
MPI_Finalize();
```

MPI Caveats

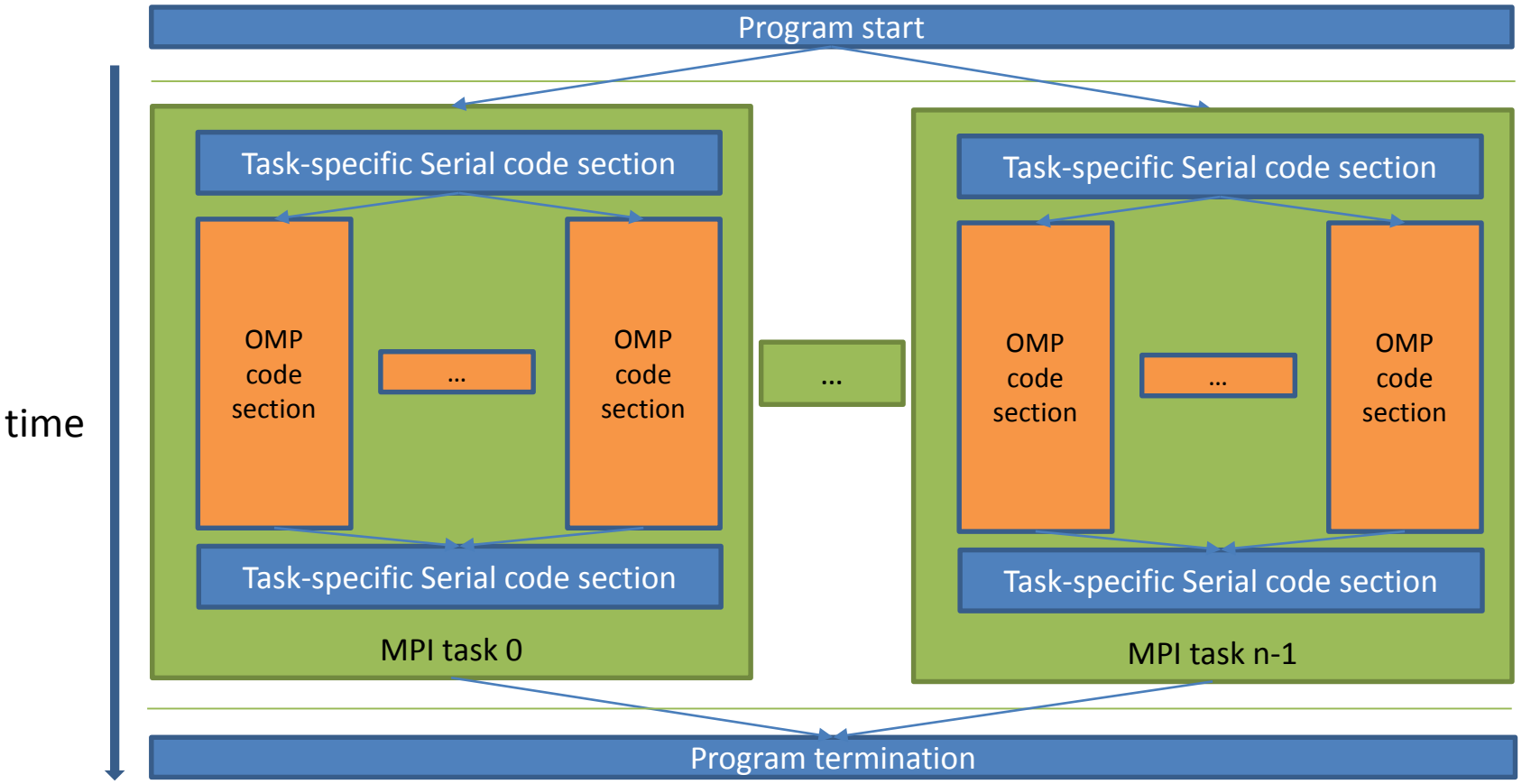
1. Deadlocks due to unmatched send/receive statements
2. Accessing wrong memory areas due to pointer usage
3. Large communication overhead
4. Network congestions (all-to-all routines) → can affect other users on an HPC system
5. … many more …

MPI is extremely simple regarding its principles, but it may become arbitrarily complex and error prone in practice.

Outline

1. Parallelising an algorithm – a real-life non-computing example
2. OpenMP: An example and some hints
3. MPI at a glance
4. *… some thoughts …*
5. Where to go from here?

MPI + OpenMP: Hybrid Code



Is Hybrid Programming a Must or an Option?

Why not just put as many MPI tasks on a node as there are cores and forget about OpenMP?

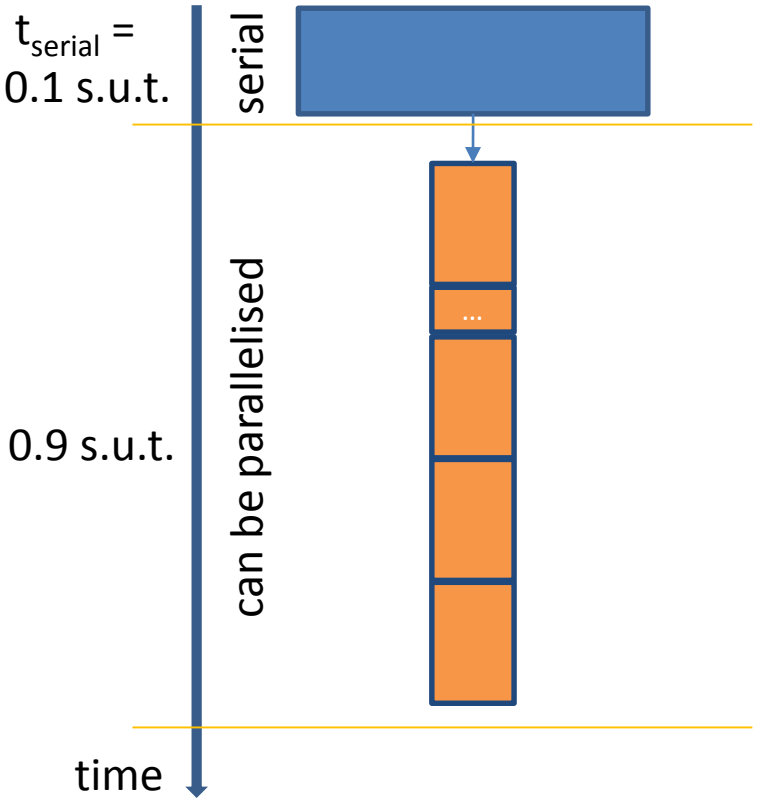
Sometimes this is reasonable and effective.

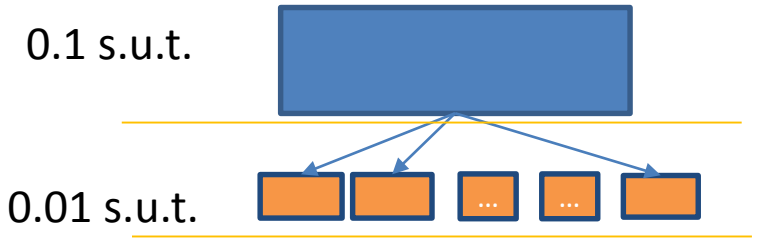
Sometimes it is not, e.g. when

1. memory requirements prevent this.
2. this would lead to too much communication.
3. OpenMP can handle low-level parallelism more efficiently than MPI.

→ You need to analyse the inner workings of your code!

Amdahl‘s Law and the Limit of Strong Scaling





Runtime cannot be reduced below t_{serial}

↓

Speed-up limited by $1/t_{serial}$

s.u.t. = serial unit time := 1 for a purely serial run

Implications of Amdahl‘s Law

1. Even if 99% of the code can be perfectly parallelised, the speed-up cannot be larger than 100. On a cluster with thousands of cores this is a serious restriction!

2. Non-parallelisable and parallelisable code sections need to be identified and the corresponding runtime fractions have to be estimated BEFORE even thinking about parallelising code.

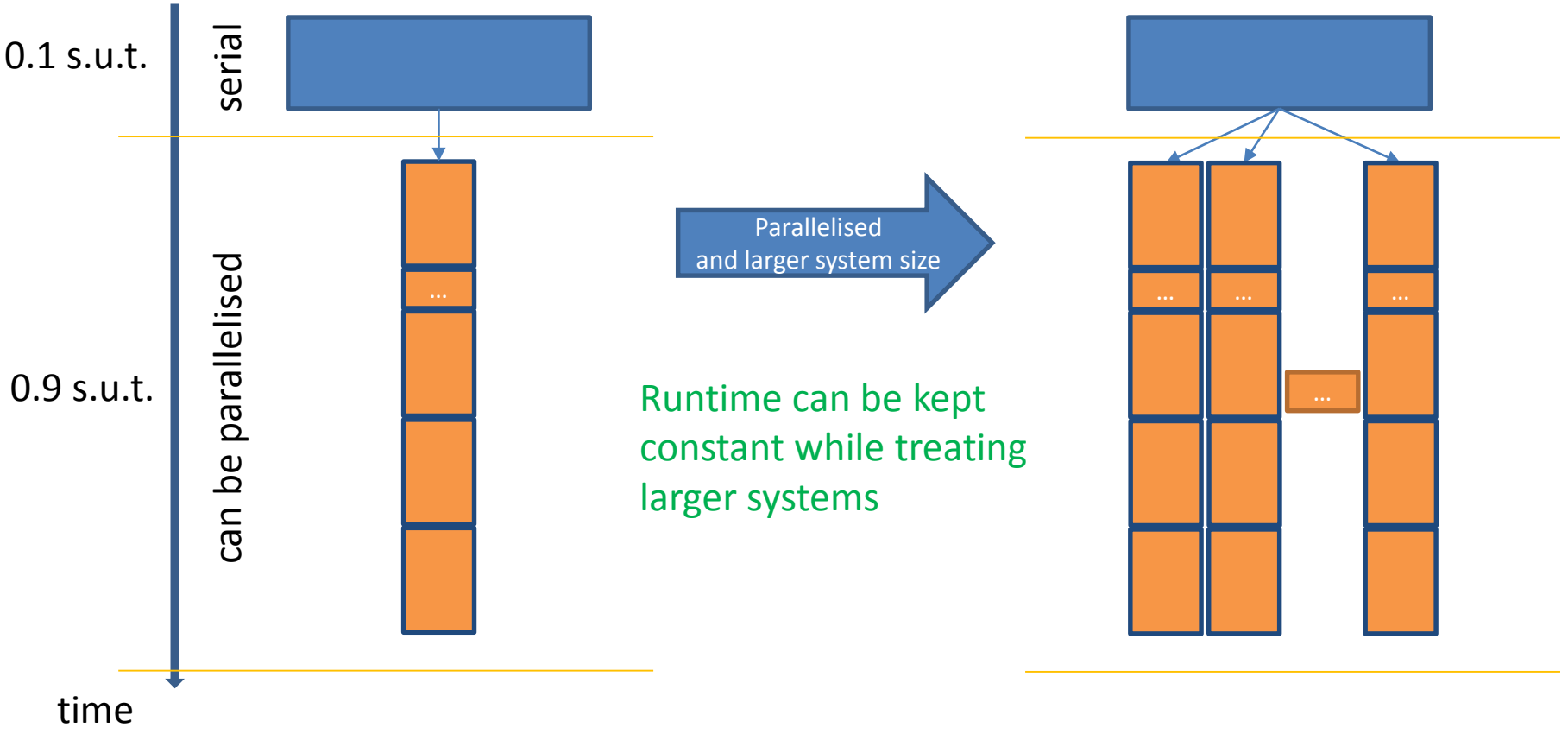
3. If parallelisation seems unreasonable: Re-think the underlying algorithm:
   Has it been thoroughly optimised?
   Could the serial section be altered to make it parallelisable?

Mind Surprises!

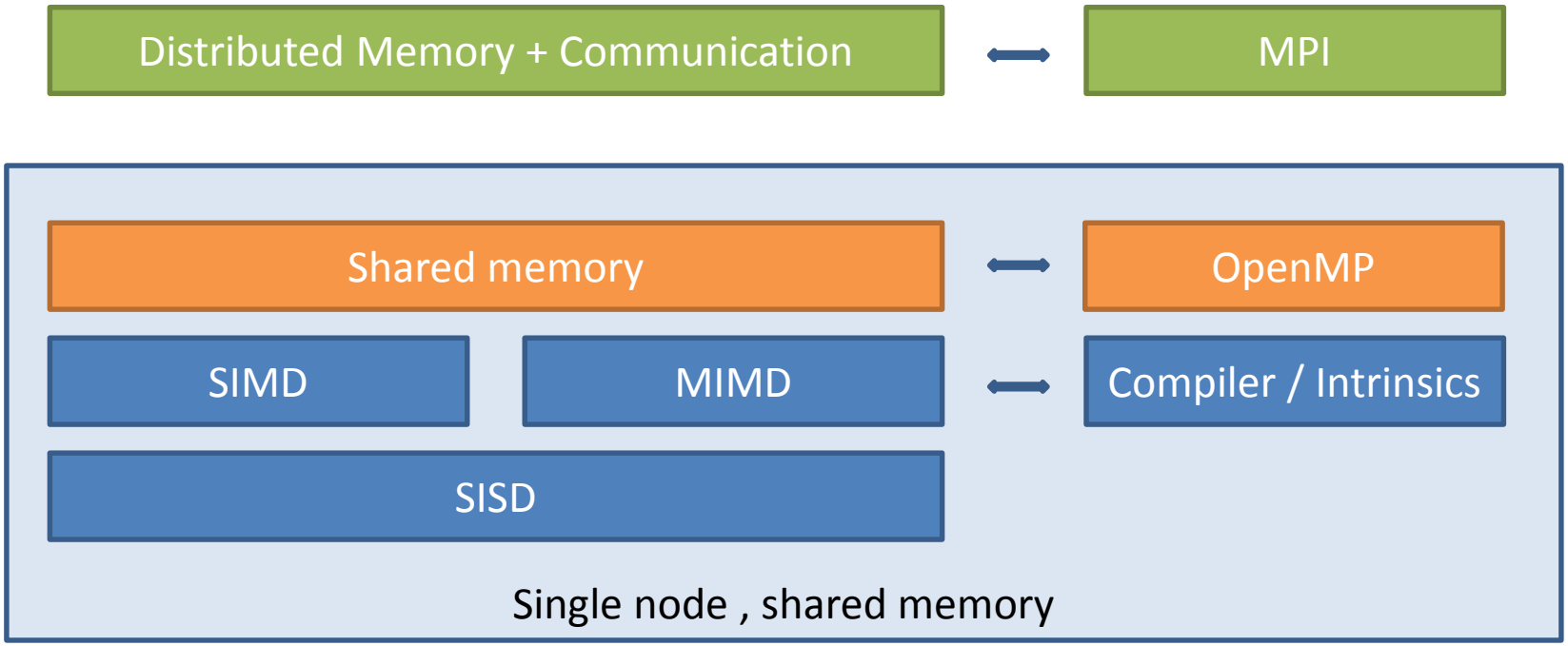
Some serial parts in the code may not be obvious, but only become evident when trying to parallelise:

1. I/O
2. Initialisation routines
3. Random number generation
4. OpenMP logistics
5. …

Amdahl‘s Law Circumvented: Weak Scaling



Hierarchy of Parallelisation



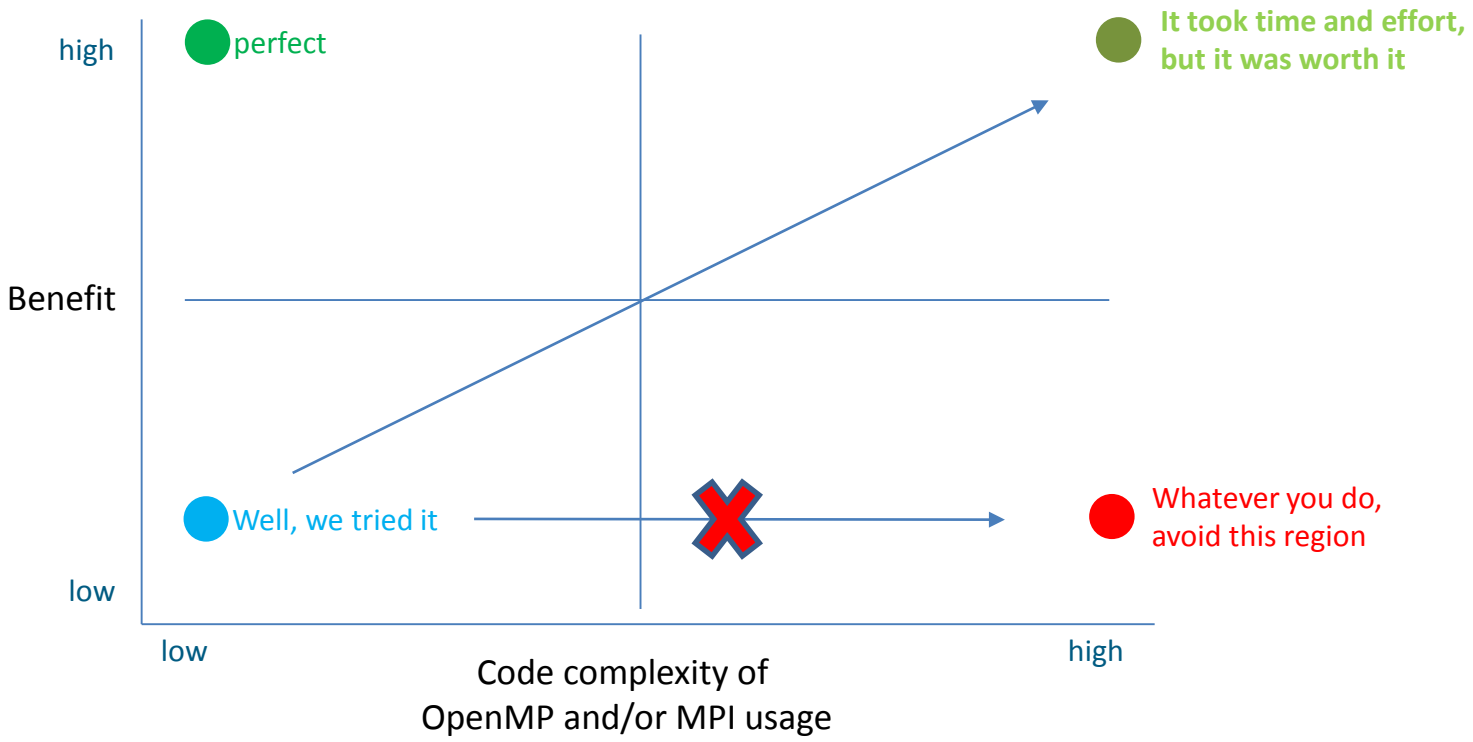
Outline

1. Parallelising an algorithm – a real-life non-computing example
2. OpenMP: An example and some hints
3. MPI at a glance
4. *… some thoughts …*
5. Where to go from here?

Suggested Path to Hybrid Parallel Code

1. Think about the algorithms you use and if there are parts that could run in parallel.
2. First, try to use MPI in the simplest possible manner and only very few tasks to try things out.
3. Check speed-up and efficiency of this code.
4. Only then you should look at possibilities to use OpenMP as well.
5. Add OpenMP and check speed-up and efficiency again.
6. Always compare results and debug the code in between!

KISS – Keep it Simple and Structured



Other Ways to Parallelise Code

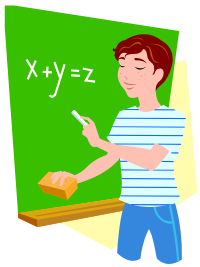
MPI and OpenMP are the most common,
but not the only ways to parallelise code.

Other options are e.g.

1. pthreads
2. vendor-specific language extensions
3. hardware-specific language extensions (gpu accelerator cards)
4. ...

You are not alone ...

The Jülich Supercomputing Center (JSC) at the Forschungszentrum Jülich offers training and education in various topics and at all levels:



→

http://www.fz-juelich.de/ias/jsc/EN/Expertise/Workshops/workshops_node.html

JÜLICH
FORSCHUNGSZENTRUM

… and remember:

DON‘T PANIC!

and CARRY a DEBUGGER

A scientist's guide to parallel programming

Florian Janetzko, Daniel Rohe and Alexander Schnurpfeil